Received: 2 October 2017 | Revised: 8 March 2018 | Accepted: 16 April 2018
DOI: 10.1002/cpe.4650



WILEY

SPECIAL ISSUE PAPER

# Domain knowledge specification for energy tuning

Madhura Kumaraswamy[1] | Anamika Chowdhury[1] | Michael Gerndt[1] | Zakaria Bendifallah[2] | Othman Bouizi[2] | Uldis Locans[2] | Lubomír Říha[3] | Ondřej Vysocký[3] | Martin Beseda[3] | Jan Zapletal[3]

[1]Faculty of Informatics, Technical University of Munich, Munich, Germany
[2]Intel ExaScale Labs, Paris, France
[3]IT4Innovations, National Supercomputing Centre, VŠB-TUO, Ostrava, Czech Republic

**Correspondence**
Madhura Kumaraswamy, Faculty of Informatics, Technical University of Munich, 80333 Munich, Germany.
Email: kumarasw@in.tum.de

**Funding information**
European Union's Horizon 2020 Programme, Grant/Award Number: 671657

## Summary

To overcome the challenges of energy consumption of HPC systems, the European Union Horizon 2020 READEX (Runtime Exploitation of Application Dynamism for Energy-efficient Exascale computing) project uses an online auto-tuning approach to improve energy efficiency of HPC applications. The READEX methodology pre-computes optimal system configurations at design-time, such as the CPU frequency, for instances of program regions and switches at runtime to the configuration given in the tuning model when the region is executed. READEX goes beyond previous approaches by exploiting dynamic changes of a region's characteristics by leveraging region and characteristic specific system configurations. While the tool suite supports an automatic approach, specifying domain knowledge such as the structure and characteristics of the application and application tuning parameters can significantly help to create a more refined tuning model. This paper presents the means available for an application expert to provide domain knowledge and presents tuning results for some benchmarks.

**KEYWORDS**

automatic tuning, domain knowledge, DVFS, energy efficiency, high performance computing

## 1 | INTRODUCTION

Energy efficiency and consumption have become the most important and challenging issues in current HPC systems and in designing future exascale computing systems.[1] To overcome these challenges, the European Union Horizon 2020 project READEX (Runtime Exploitation of Application Dynamism for Energy-efficient Exascale computing) aims to deliver the first standalone auto-tuning framework that dynamically tunes large-scale HPC applications.[2] Unlike previous works focusing on static tuning,[3] READEX applies a dynamic tuning approach by switching tuning parameters during application execution. Optimized *system configurations* can thus be configured for dynamic variation in characteristics of the program's execution, such as the compute intensity. Thus, the program's potential for energy reduction is significantly increased.

READEX leverages the hardware features for *Dynamic Voltage and Frequency Scaling (DVFS)* of the Intel Haswell processor family that allows the userspace governor to set the frequency for individual cores as opposed to full sockets as seen in previous processor lines. In addition, the frequency of the uncore chip area can be tuned in the Haswell processor independent of the core frequency.[4] The frequencies can be controlled by setting machine specific registers (MSRs).

The READEX methodology is a two-stage approach and consists of *Design Time Analysis (DTA)* and *Runtime Application Tuning (RAT)*. DTA applies an auto-tuning approach to compute best system configurations for instances of program regions, called *runtime situations (rts's)*. Different runtime situations of a region are identified in the READEX methodology by *region identifiers*, eg, the call path of the given instance. The found best system configurations are written to the *tuning model*, which is then read during production runs for Runtime Application Tuning. During runtime tuning, the best system configurations are dynamically switched for rts's. This switching occurs independently in individual application processes.

Before DTA, the application is first analyzed for its *tuning potential* (*pre-analysis step*), ie, the inherent dynamism in the execution characteristics with the potential for energy reduction by dynamically switching system configurations at runtime.

Pre-analysis first filters out regions with too short execution time to prevent excessive measurement overheads. Coarse granular regions that constitute most of the execution time are selected for dynamic tuning and are identified as *significant regions*[5] using a READEX tool called readex-dyn-detect. `readex-dyn-detect` then computes the tuning potential of significant regions for dynamic tuning.

If significant tuning potential was found in the pre-analysis step, DTA determines the application's tuning model. READEX targets applications that exhibit an iterative behavior in the form of a main progress loop, which is called a *phase region*, and whose individual time steps are called *phases*. DTA exploits this iterative structure in its auto-tuning approach. Within a single execution of the application, different system configurations are assessed in different program phases. This results in best configurations for the rts's. Finally, the rts's are clustered according to their best configurations, and the information is written to the tuning model file.

This tuning model file then guides the Runtime Application Tuning. At the start of a region, first, the rts are determined based on the observed region identifiers, and the best configuration is taken from the tuning model. The current system configuration is then switched to the recommended one.

The implementation of DTA is based on the Periscope Tuning Framework (PTF)[6] and the Score-P[7,8] monitoring system. Runtime Application Tuning is implemented as an extension of Score-P called the READEX Runtime Library (RRL).

This paper presents new features of the READEX tool suite, enabling the application developer to specify domain knowledge that is used in DTA and runtime tuning for additional application tuning. This domain knowledge covers additional *identifiers* for the application structure and characteristics and application-level tuning parameters, such as the types of solvers, access strides, or number of subdomains for exploiting the dynamicity. READEX uses the so-called identifiers to predict at runtime the characteristics of an upcoming rts. The READEX *Domain Knowledge Specification Interface* supports region identifiers to distinguish rts's, phase identifiers to distinguish phase characteristics, and input identifiers to distinguish executions with different application inputs. Without these identifiers, rts's of a significant region cannot be distinguished during runtime tuning, and thus would end up with the same system configuration even if they have a different behavior. Hence, these identifiers will improve the tuning model by distinguishing rts's and assigning them to different system configurations.

Section 2 first gives a general overview about the implementation of DTA and Section 3 outlines related work. Section 4 then presents the integration of the domain knowledge specification in the design-time workflow. It describes how domain knowledge, such as the program regions and the phase region (Section 4.1.1), application characteristics (Section 4.2), and the ATP (Section 4.3) specifications can be added by the application expert. Section 5 provides details on the implementation of the identifiers and the ATPs and how DTA uses the domain knowledge to generate a more refined tuning model. The static and dynamic energy savings obtained after applying region identifiers for the Multi-Grid (MG) benchmark from the NAS parallel benchmark suite[9] are discussed in Section 6.1, followed by a description on the trend in the energy consumption for the multi-zone version of the Block Tri-diagonal solver (BT) for different input sets in Section 6.2. An evaluation of the results obtained for the ESPRESO library[10] using preconditioners, domain decomposition, and iterative solver switching as the Application Tuning Parameters is also presented in Section 6.3. In the end, this paper will summarize the important results and draw the concluding remarks.

## 2 | DESIGN-TIME ANALYSIS

DTA is implemented using the Periscope Tuning Framework (PTF) that was developed at the Technical University of Munich, Germany. PTF is a distributed framework consisting of the frontend, the tuning plugins, the experiment execution engine, and a hierarchy of analysis agents.[6] PTF provides several tuning plugins for different tuning aspects, such as energy consumption and compiler flags. A tuning plugin determines a number of system configurations based on expert knowledge and then assesses the configurations via experiments that measure the objective function.

A novel tuning plugin called the READEX Tuning Plugin was developed for PTF to perform DTA by running experiments that currently evaluate three tuning parameters, ie, CPU frequency, uncore frequency, and the number of OpenMP threads within a single program run. Each experiment is an execution of the phase region. First, the plugin reads the ranges (minimum, maximum, and the step size) of the tuning parameters, the search algorithm, the objective to tune the application for, and the *Application Tuning Parameters* (ATPs). The plugin uses one of several predefined search algorithms to walk the search space consisting of the cross-product of the tuning parameters and ATPs.

For each selected configuration, an experiment is executed in which the effect of the system configuration on the rts's of significant regions are measured. The measurements for the phase as well as the objective values for the rts's are retrieved from Score-P and propagated to the tuning plugin. The best system configuration for the rts's of the significant regions is returned for the selected objective. The result of this search, including the objective values, the execution time and the number of instances, as well as the best found configuration for each rts are then stored in an RTS database.

Finally, a *classifier* groups the rts's with similar or identical best found configurations into *scenarios* using a similarity score that is computed by aggregating the closeness of system configurations. The clustering limits the number of scenarios and thus reduces the associated runtime overhead. A *selector* then chooses the best configuration for each scenario for the specified objective. The knowledge obtained during DTA, such as the best-found system configurations for individual scenarios is encapsulated in a tuning model (TM).

For production runs, the tuning model is forwarded to the READEX Runtime Library (RRL), which performs runtime tuning by detecting the scenarios and dynamically switching to the best configurations for upcoming rts's.

## 3 | RELATED WORK

There are limited numbers of automatic tuning approaches and tools available for runtime optimizations using domain knowledge. Here, we present the most recent auto-tuning approaches used in research projects.

The AutoTune project[6] developed a static tuning approach based on Dynamic Voltage and Frequency Scaling (DVFS), which involves increasing or decreasing both the voltage and the frequency. During this project, a specialized tuning plugin was implemented using the energy consumption as the tuning aspect. The plugin generates models using the performance data of a certain platform to predict the energy consumption, time, and power at different CPU frequencies. The plugin uses the *enopt* library[11] to configure the CPU frequency for different regions and to collect energy measurements. The library updates a timer counter every 10 seconds to collect energy measurements. This is a static tuning approach where the CPU frequency remain unchanged for a whole program region. In the READEX approach, both the CPU frequency and uncore frequency can be configured dynamically for all the program regions.

The MATE[12] framework implemented a tuning environment to tune applications dynamically during runtime. Here, the application monitoring takes place as well as detecting performance bottlenecks. Then, the application is adapted dynamically due to the behavior changes without recompiling or re-running. The tuning is driven by aspect-specific tuning components. For example, in master/worker type applications, the number of workers is adapted to the number of tasks. However, the framework only limits the tuning approach to MPI processes. In contrast to READEX, the MATE framework focuses on performance improvements. It is the responsibility of the expert to provide the aspect-specific tuning components. READEX provides a general tuning strategy independent of the application structure.

Calore et al[13] used a static Dynamic Voltage and Frequency Scaling (DVFS) to tune modern processors, namely, an NVIDIA K80 Graphics Processing Unit (GPU) and an Intel Haswell CPU. They target the energy costs of HPC applications by tuning CPU and GPU frequencies. In this work, they focused on a hardware tuning approach. Although they considered a few software tools for energy optimization, they preferred to avoid the software approach, and thus could not exploit, eg, application tuning parameters.

The ANTAREX project[14] uses a domain-specific language (DSL) for tuning applications targeting energy-efficiency on multi-core CPUs and accelerators. This DSL approach uses adaptive and auto-tuning strategies to enforce runtime application tuning. The DSL template interacts with the runtime resource and power manager to configure software tuning parameters (such as application parameters, code transformations, and code variants) for the application regions. The additional compilation step is required to translate from the DSL template into the target language. This DSL approach is imposed only on ARM-based multi-cores and GPGPUs, while the READEX tuning approach targets different HPC platforms. Furthermore, READEX targets standard HPC applications, while ANTAREX requires applications to be programmed in the DSL.

The use of domain knowledge in the READEX methodology for energy efficiency significantly extends our current dynamic tuning approach. None of the aforementioned works have yet explored this aspect to the best of our knowledge.

## 4 | DOMAIN KNOWLEDGE SPECIFICATION

The READEX programming paradigm developed the *Domain-level Knowledge Specification Interface* in order to enhance the tuning process. This is done by enabling the application expert to provide user-level specifications to expose domain-level knowledge, such as information characterizing application dynamism and the parameters to be used for tuning. The specification allows the identification of new system scenarios, and covers the following aspects of domain knowledge:

1. Specification of application structure (Section 4.1),
2. Specification of application characteristics (Section 4.2), and
3. Specification of application tuning parameters (Section 4.3).

The three aspects of the domain knowledge specification are integrated at different stages of the DTA workflow, as shown in Figure 1. As the first step, the application expert specifies the application structure, which includes the phase region and additional program regions that should be considered for tuning. Then, the application dynamism is detected with the help of a tool called `readex-dyn-detect`. Based on the result of the dynamism detection, READEX tuning is either stopped in case of no available dynamism, or the application expert may identify application characteristics to support DTA in case there is enough dynamism. These characteristics are marked by identifiers, such as region identifiers, phase identifiers, and input identifiers. The identifiers will allow DTA to generate a more sophisticated application tuning model by using a specific setting of the tuning parameters for certain application characteristics. While the region and phase identifiers are specified in the source code via Score-P macros, input identifiers are added in files accompanying the original input file.

Finally, the Application Tuning Parameters (ATPs) are specified in the application source files via Score-P macros calls to the ATP library. When the application is executed, the ATPs are written to the ATP description file and are then read by PTF at the start of the analysis. Separate tuning model files are generated for each input specification, which are then merged into the final application tuning model. DTA terminates after processing all the inputs. The generated merged tuning model is then read during the runtime tuning stage.

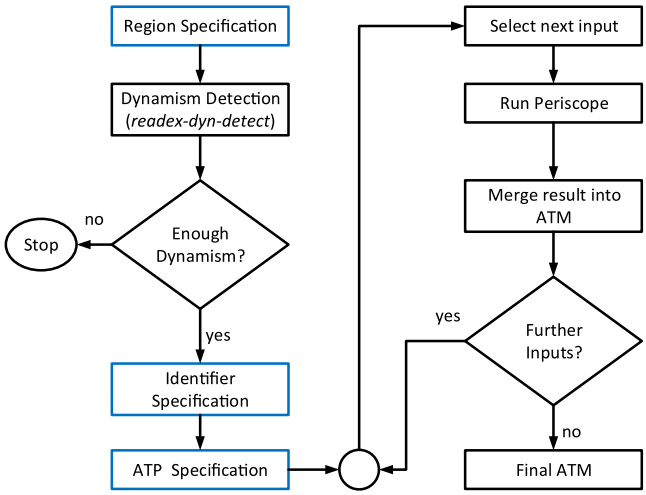


**FIGURE 1** Specification of the aspects of domain knowledge at different stages of the DTA workflow

## 4.1 | Specification of application structure

### 4.1.1 | Region and phase specification

The READEX approach is based on tuning program regions. The decomposition of the program into program regions is typically defined by the syntax of the programming language. Common types of regions are subroutines, loops, and structured blocks. Besides standard regions, the application expert might be able to identify additional regions that are not represented as a standard region but may have interesting characteristics or tuning potential. Typical use-cases for user regions are as follows.

- User regions can combine several calls to different functions, which belong together and are too fine granular for switching the configuration individually.
- They can identify certain parts of an algorithm that would otherwise not be a target of tuning because this part is not represented by a standard region type of the programming language.
- If instrumentation is too fine granular and leads to a lot of overhead, automatic instrumentation can be switched off and significant regions can be manually instrumented.

The READEX tuning approach targets applications that have a central progress loop, called a phase region that iteratively performs some computation. The phase region is a program region that defines the phases of execution. Although this loop can be implemented by a standard loop construct of the programming language, automatic detection of the phase region is difficult. Score-P[7] offers an *online access phase region* to enable external tools to configure Score-P dynamically when a phase is started. This configuration mechanism is used in DTA to perform experiments for evaluating different system configurations. Both the start and the end of the phase region entail a barrier synchronization of all participating processes when an online tool like PTF is connected to Score-P.

The phase region and user regions are defined in READEX using Score-P macros. These macros can enclose arbitrary code and are instrumented automatically. As a result, the annotated regions can be handled by the READEX tool suite like any other program region. First, the region handles are defined with the macro `SCOREP_USER_REGION_DEFINE`. Then, the start and end of the user region are marked with the macros `SCOREP_USER_REGION_BEGIN` and `SCOREP_USER_REGION_END`, while the phase region is enclosed in the macros `SCOREP_USER_OA_PHASE_BEGIN` and `SCOREP_USER_OA_PHASE_END`.

#### Use-Case

A use-case for the phase region and a user region specification for the MG (MultiGrid) benchmark from the NAS parallel benchmark suite[9] is shown in Listing 1. MG uses a V-cycle to solve a discrete Poisson equation on a 3D grid. It is based on a hierarchy of grid levels, where the maximum level is the finest grid with the highest resolution. After computing an initial residual, the iteration loop of MG executes an entire V-cycle starting from the highest grid level. First, the result on the current grid level $k$ is projected to the next coarser grid level $k-1$. When the coarsest grid is reached, an approximate solution is computed. The result is then interpolated from the coarser to the finer grid, where the residual is calculated and a smoother is applied to correct the result. Finally, the result is then propagated to the finest grid.

Several V-cycles are executed to solve the equation. The body of the iteration loop of MG is marked as the phase region, as shown in lines 6 and 10 of Listing 1, and thus can be used to, eg, assess different candidate configurations during DTA. A user region is defined for the subroutine `interpolate`, as shown in lines 38 and 43 of Listing 1.

```
1  #include "SCOREP_User.inc"
2  SCOREP_USER_REGION_DEFINE(R1)
3   ...
4  do it = 1, max_iter
5    ! Phase region begins
6    SCOREP_OA_PHASE_BEGIN(R1, "VCycle", 0)
7    ! Execute a V-Cycle
8    call mg3P (...)
9     ...
10   SCOREP_OA_PHASE_END(R1)
11   ! Phase region ends
12 enddo
13  ...
14
15 subroutine mg3P (...)
16   do k = max_level, min_level+1
17     ! Restrict residual from the fine grid to the coarser grid
18     call rprj3 (...)
19   enddo
20
21   ! Compute an approximate solution on the coarsest grid
22   call psinv (...)
23   do k = min_level+1, max_level-1
24     ! Interpolate the result from coarser to finer grid
25     call interpolate (... , k)
26     ! Compute the residual for grid level k
27     call resid (...)
28     ! Apply the smoother to correct the error
29     call psinv (...)
30   enddo
31    ...
32 end subroutine mg3P
33
34 ! Interpolate to grid level k
35 subroutine interpolate (... , k)
36   SCOREP_USER_REGION_DEFINE(R2)
37   ! User region begins
38   SCOREP_USER_REGION_BEGIN(R2, "interp",0)
39   SCOREP_USER_PARAMETER_DEFINE(intParam)
40   ! Region identifier for grid level k
41   SCOREP_USER_PARAMETER_INT64(intParam,"level",k)
42    ...
43   SCOREP_USER_REGION_END(R2)
44   ! User region ends
45   end subroutine interpolate
```

**LISTING 1** Phase region, user region, and region identifier specifications for the MG benchmark

## 4.2 | Specification of application characteristics

### 4.2.1 | Region identifier specification

Region identifiers, which may be specified by the application expert, along with the region name and the call path, can be used to distinguish runtime situations with different characteristics. Without these identifiers to help the DTA identify and distinguish runtime situations, the tuning model cannot specify different configurations. Region identifiers are specified as Score-P user parameters and can be of type integer or string. They are defined using the Score-P macros `SCOREP_USER_PARAMETER_INT64(handle,name,value)` and `SCOREP_USER_PARAMETER_STRING(handle,name,value)`, respectively. The parameters determine region identifiers for a program region. In FORTRAN, first, a handle for each parameter has to be declared with a `SCOREP_USER_PARAMETER_DEFINE` macro. Then, the handle is used to associate a name and a value with the parameter.

#### Use-Case

In Listing 1, the size of the grid processed in the call to `interpolate(..., k)` in the MG benchmark gets higher when going from the minimum grid level to the maximum. At a certain grid level, the computation switches from being compute bound to memory bound. To enable DTA to determine special system configurations for compute and memory bound runtime situations, the application expert can add a region identifier for the grid level inside the `interpolate` region. First, the identifier is defined in line 39 of Listing 1, and then associated with the value of the grid level, as shown in line 41.

### 4.2.2 | Phase identifier specification

Phase identifiers identify phases with different characteristics or behavior and can be used in the tuning model to distinguish rts's based on the variation in the behavior of the phase. Phases that have similar characteristics can be grouped together into a cluster using phase identifiers, such as the degree of sparsity of the matrix or the arithmetic intensity of the phase region, thus enabling the prediction or selection of different best configurations for different phase clusters. Moreover, different configurations can be set for rts's in different phase clusters.

Phase identifiers are provided in the same way as region identifiers (Section 4.2.1) via Score-P user parameters that are attached to the phase region. Phase identifiers should be chosen carefully as they have a high impact in selecting the best configuration for the phases in a cluster. Typically, these should be provided by the application expert who knows how the application behaves when certain aspects in the code are modified.

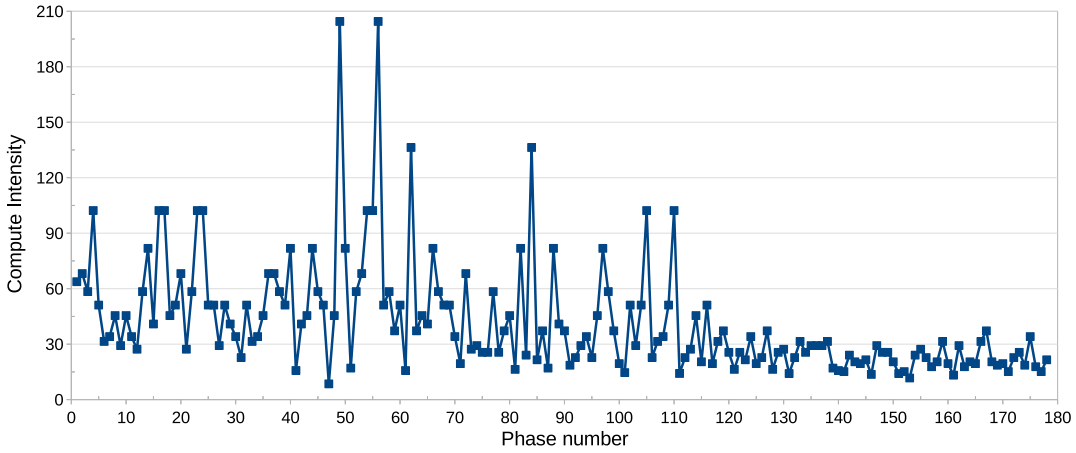


**FIGURE 2** Variation of the compute intensity ($\frac{\text{Total Instructions Retired}}{\text{L3 Cache Misses}}$) to the phase number of the time loop in INDEED

### Use-Case

DTA was applied to the INDEED code after the addition of the phase and region identifiers. INDEED[15] is a sheet metal forming simulation software with an implicit time integration. It uses an adaptive mesh refinement, and the number of finite element nodes that it uses increases with every time step, resulting in an increasing computational cost.

The time loop of INDEED computes the solution to a system of equations until equilibrium is reached for every change in the number of contact points between each tool and the workpiece. The time loop thus varies in its behavior and was annotated as the phase region. Figure 2 indicates that the compute intensity, which is derived by $\frac{\text{Total Instructions Retired}}{\text{L3 Cache Misses}}$ varies with the phase number. Hence, the application is dynamic, and the dynamism arises due to increasingly refined regions of the workpiece making contact with the tool as the iterations progress.

Phase identifiers such as the *Compute Intensity* and the *Conditional Branch Instructions* enable DTA to cluster phases that have a similar behavior into a group. Hence, phases that have a higher compute intensity depicted by the spikes in Figure 2 may be grouped into one cluster. DTA may now use the phase identifiers to select a different best configuration for each cluster for the specified objective.

## 4.2.3 | Input identifier specification

The READEX approach characterizes variations in application executions for different input sets via input identifiers provided by the application expert. As a result, DTA will be able to identify more rts's with different characteristics, which will eventually improve the tuning model. Input identifiers may be simple, such as the grid size of the application domain and the number of processors, or complex, like the number of contact points in metal forming simulations like INDEED.

The Domain Knowledge Specification Interface allows to provide input identifiers in an accompanying input specification file in the form of key-value pairs, as shown in lines 2 and 3 of Listing 2. The input identifier specification files are interpreted by both PTF during DTA and the RRL during runtime tuning. PTF reads the files via the command line option and RRL interprets them via an environment variable.

In the MG benchmark, while the grid level switches from a coarser grid to the next finer grid, the compute intensity characteristics might change, eg, from memory bound to compute bound. On which grid level this switch happens, depends, eg, on the grid size given in the input file. Thus, the region identifier for the grid level *k* (Section 4.2.1) for the rts's of the regions is not sufficient to improve the tuning model for different input sizes. As the optimal configuration is dependent on the size of the finest grid, this grid size has to be taken into account when selecting the configuration for a certain grid level at runtime. In addition to the size of the finest grid, the number of processes is also important. The more processes are used, the better the data distributes over the caches and the earlier; in terms of grid level, the application switches between memory and compute bound. The numbers of processes and threads are considered as standard input identifiers and need not be given in the specification file.

```
1 identifier name     : <value>
2 ContactPoints  :   <simple, complex>
3 ProblemSize    :   <256 256 256>
```

**LISTING 2** Input identifier specification file IID_SPEC

## 4.3 | Application Tuning Parameters

In addition to hardware and system tuning parameters, READEX also targets application level parameters, ie, parts of the code itself, which could be used as tuning parameters. The simplest example of this is the case where different implementations of the same algorithm are available, each having its own impact on performance and energy. Some examples of application tuning parameters include, ie, choosing between different decomposition algorithms, preconditioners, or also blocking factors. The aim of using Application Tuning Parameters (ATPs) is to exploit the possibility to switch between the different implementations or, in a more general sense, the possibility for READEX to choose between code level alternatives at runtime.

In order to optimize energy efficiency by choosing the best settings for ATPs, the application developer needs to pass the knowledge on to the tuning system. READEX provides an annotation API, ie, the source code is annotated at specific locations to provide the tuning system with the necessary details on the tuning parameters.

The *ATP component* ensures support for ATP handling in the READEX tool suite. The overall process by which ATPs are handled can be summarized by the following steps.

1. *Annotation and instrumentation*: The annotation step is primarily done by the application developer. It consists of exposing *control variables*, which drive the switching between different possible alternatives by marking them with API functions as ATPs. The marking allows to provide additional details, such as the range of values to the tuning system at runtime. All API functions are implemented in the *ATP library*.
2. *Design-Time Analysis*: During the first phase of the application in DTA, the ATP library generates the information about ATPs into file, and the ATP component provides valid settings for the ATPs to the READEX tuning plugin in PTF. During an experiment, the ATP library will retrieve a specified setting and assign the value to the control variable or assign the default value that is to be given with the ATP specification.
3. *Runtime Application Tuning*: In a production run, the behavior of the ATP library changes. The annotation part (API parts involved in parameter detail gathering) is shutdown, thus leaving only the part related to value assignment active. Hence, the ATP component receives the values from the RRL and assigns them to the control variables.

### 4.3.1 | ATP declaration and retrieval

The control variables used for ATPs can be of simple data types, such as integers, booleans or floats, as well as complex types such as arrays or structures. The current version of the READEX tool suite is set to handle *integer* data types, as exploring integer ranges and solving constraints between them requires a relatively reasonable amount of compute power compared to floats and complex structures. Hence, in the cases where a tuning parameter is a complex structure, the user may, when possible, simplify it by mapping its possible values to integer variables.

Listing 3 presents the relevant API functions in the ATP library. The first set allows to declare ATPs and their properties to the READEX tuning system. For this task, two API functions must be used jointly. The first, *ATP_PARAM_DECLARE()*, is used to declare the parameter name, type, default value, and the domain (line 1). The second, *ATP_PARAM_ADD_VALUES()* (line 2), allows to supply the possible values the parameter can take. If the ATP was declared of type *RANGE*, this function will provide an array of three values (minimum value, maximum value, and an increment); otherwise for type *ENUMERATION*, it provides an array of all the possible values the parameter may take. Both functions are only active during the first phase of the application in DTA to gather parameter details.

During the experiments in DTA and during runtime tuning, ATP values have to be assigned to the corresponding control variable. The *ATP_PARAM_GET()* function (line 3) makes the mapping between the declared ATP and the corresponding control variable.

### 4.3.2 | Domains

An application source code may contain several ATPs. Ideally, these would be independent from each other, enabling independent tuning. In practice, this is not always the case; the values of a parameter may depend on those of another one, and hence, this would translate into the notion of constraints between parameters. Therefore, the application developer may indicate to READEX that a number of parameters have constraints between them by putting them in the same domain.

In the API, domain declaration is included within the parameter declaration call, where both parameter name and domain name need to be supplied. In addition, if no domain name is supplied, the parameter is assigned to the *default* domain.

```
1 ATP_PARAM_DECLARE(const char *param_name, const char param_type, int default_value, const char *domain_name)
2 ATP_PARAM_ADD_VALUES(const char *param_name, void *vArray, int array_size, const char *domain_name)
3 ------------------------------------------------------------------------------------------------------------
4 ATP_PARAM_GET(const char *param_name, void *atp_address, const char *domain_name)
5 ------------------------------------------------------------------------------------------------------------
6 ATP_CONSTRAINT_DECLARE(const char *constraint_name, const char *constraint_expr, const char *domain_name)
7 ATP_EXPLORATION_DECLARE(const char *explorations_list, const char *domain_name)
```

**LISTING 3** ATP library API functions

```
Let mesh, solver be integers
  solver ranges in [1,5] with an increment of 1
  mesh   ranges in [0,80] with an increment of 10
constraints:
  1. if solver equals one then mesh must be between 0 and 40.
  2. if solver equals two then mesh must be between 40 and 80.
  3. if solver is above two then mesh must be equal to 120.
```

**FIGURE 3** Example showing the dependence of the number of mesh points on the solver to be used

```
1  void foo(){
2    int atp_cv;
3     ...
4    ATP_PARAM_DECLARE("solver", RANGE, 1, "DOM1");
5    int32_t solver_values[3] = {1,5,1};
6    ATP_ADD_VALUES("solver", solver_values, 3, "DOM1");
7    ATP_PARAM_GET("solver", &atp_cv, "DOM1");
8
9    switch (atp_cv){
10     case 1:
11          // choose solver 1
12          break;
13     case 2:
14          // choose solver 2
15          break;
16      ...
17   }
18   int32_t hint_array = {GENETIC, RANDOM};
19   ATP_EXPLORATION_DECLARE(hint_array, "DOM1");
20 }
21
```

```
22 void bar(){
23   int atp_ms;
24    ...
25   ATP_PARAM_DECLARE("mesh", RANGE, 40, "DOM1");
26   int32_t mesh_values[3] = {0,80,10};
27   ATP_ADD_VALUES("mesh", mesh_values, 3, "DOM1");
28   ATP_PARAM_GET("mesh", &atp_ms, "DOM1");
29   ATP_CONSTRAINT_DECLARE("const1", "(solver = 1 && 0 <=
         mesh <= 40) || (solver = 2 && 40 <= mesh <= 80) || (solver
         > 2 && mesh = 120)", "DOM1");
30   if ((atp_ms > 1) && (atp_ms <= 40)){
31        // choose mesh size 1
32   }
33   if ((atp_ms > 40) && (atp_ms <= 80)){
34        // choose mesh size 2
35   }
36   if (atp_ms == 120){
37        // choose mesh size 3
38   }
39 }
```

**LISTING 4** ATP constraint and exploration declaration with the ATP library

### 4.3.3 | Constraints

In the case where two or more ATPs have constraints between them, it is necessary to express the constraints to READEX, as this would allow to generate only valid values for the tuple of parameters. In order to take dependences between ATPs into account, the ATP description formalism allows these dependences to be expressed mathematically in the form of logical constraints. An example of such constraints is shown in Figure 3. Here, the ATPs `solver` and `mesh` are tied by the constraints 1, 2, and 3. The logical expression that can be derived from it and which READEX understands is

```
(solver = 1 && 0 <= mesh <= 40) || (solver = 2 && 40 <= mesh <= 80) || (solver > 2 && mesh = 120).
```

Constraint declaration goes through a single API function call, as illustrated in line 6 of Listing 3, where the constraint is given in the form of a logical expression. Line 29 in Listing 4 illustrates the declaration of a constraint between the ATPs `solver` and `mesh`.

In order to avoid increased complexity in constraint solving, READEX accepts affine function based constraints only. These are sufficient to handle a wide range of constraints. Therefore, in addition to logical operators, addition, subtraction, multiplication, and division are accepted to form affine functions.

### 4.3.4 | Exploration

The ATP library API provides a function call for the application developer to give hints to the READEX tuning system about what search algorithm to use in the DTA, eg, exhaustive, individual, random, or genetic. The goal being to let the developer add knowledge about the search space. The call allows to give an ordered list of exploration heuristics to DTA. It should be noted that the hints are tied to a domain, which means that all the parameters included in the domain would be subject to the same hints. Line 7 of Listing 3 shows the prototype of the exploration API call.

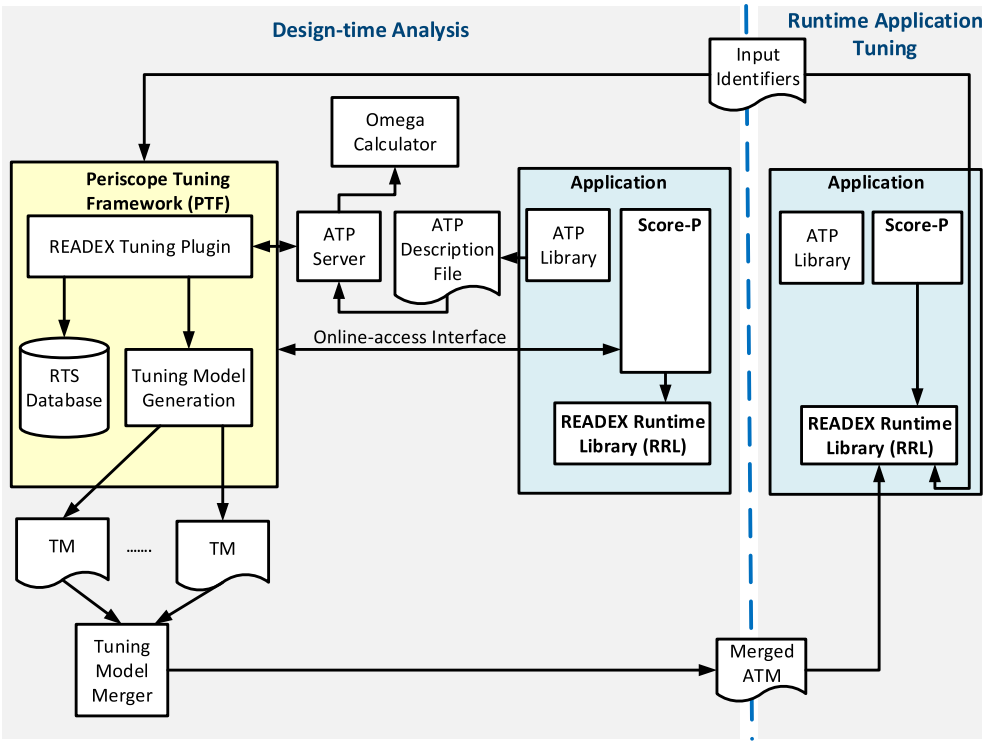


**FIGURE 4** An overview of the interaction between the domain knowledge components with other READEX components during DTA and Runtime Application Tuning. The application component occurs twice since it is run during both DTA and runtime tuning but has different interactions during the two runs

### Use-Case

Listing 4 illustrates how the ATP library API functions can be used to declare control variables from the code as application tuning parameters. It also shows how constraints between these variables can be declared.

Lines 4 to 7 and 25 to 29 in Listing 4 provide examples of ATP declaration and value retrieval where the ATP *mesh* is declared and tied to the control variable `atp_cv`. The same is done for the ATP *solver* with the control variable *atp_ms*.

## 5 | IMPLEMENTATION

First, the domain knowledge specification is added to the application to specify the phase region, as shown in Figure 1. Pre-analysis steps are then performed by `readex-dyn-detect`, which returns the significant regions and computes the tuning potential at the end of the dynamism analysis step. `readex-dyn-detect` stores the dynamism information for each significant region along with tags to specify the objectives, the tuning parameters, and the search algorithm in a READEX configuration file.

Figure 4 gives an overview of the interaction of the components of the READEX methodology with the domain knowledge during DTA and runtime tuning. The components are described as follows.

**Score-P** After the pre-analysis steps, the significant regions are annotated using Score-P macros (Section 4.1.1), and if needed, region identifiers are specified (Section 4.2.1). Score-P provides a measurement infrastructure for profiling, event trace recording, and online analysis of HPC applications. The Periscope Tuning Framework (PTF) then executes one phase of the application and requests for objective values with the default setting of the tuning parameters. The measurements for each rts are returned to the PTF analysis agents from Score-P via the Online-Access Interface. The Online Access Interface allows online tools like PTF to connect to the Score-P monitoring infrastructure linked to the application processes, when the phase region is touched the first time. The important program regions are then stored in PTF.

**ATP Library** The ATP library provides an annotation API to the application expert. The API allows to mark specific chosen control variables in the source code as application parameters. It also provides the instrumentation interface, which allows to assign the parameter values generated by PTF and prepared by the runtime library to the right control variable. At the end of the first phase of the application execution, the ATP library generates the *ATP description file* in the JSON format, which consists of details about the declared application parameters.

**ATP Server** In contrast to the ATP library, which is tied to the application itself, the ATP server is launched by PTF and reads the contents of the ATP description file. The primary task of the ATP server is to respond to PTF requests on ATPs. Some requests are simple, such as querying the list of ATP parameters, and the server can get the information by looking in the content of the ATP description file. Other requests are more complex, such as the generation of a list of valid points for the parameters, which require the resolution of the constraints held between parameters. These require the server to query a third party constraint solver software called the *Omega Calculator*.[16]

**Omega Calculator** The Omega Calculator is an affine functions constraint solver and generates the valid values for each ATP for the recorded constraints by filtering the tuning parameter values that do not satisfy the constraints. The Omega Calculator software is composed of a library *Omega Library*, which constitutes the core of the solver as well as a text interpreter to query the library. The software is registered under the BSD license, and the source code can be downloaded freely from Github.[17] One big advantage of using the Omega Calculator is the small computational time needed to solve the affine function based constraints, which makes it fit for use to solve the constraints at runtime.

**READEX Tuning Plugin** PTF performs DTA by executing the READEX tuning plugin, which reads from the READEX configuration file the objective(s) (Energy, CPU Energy, Execution Time, or Total Cost of Ownership), the tuning parameters (CPU frequency, uncore frequency and the number of OpenMP threads), the search strategy (exhaustive, random, individual, or genetic), and the significant regions. It also reads the input identifiers from the input specification file and the ATPs from the generated ATP description file. The READEX tuning plugin starts the *ATP Server* and sends a *getatpspecs* request for the ATP specification details of the application. The ATP Server responds with the number of application tuning parameters and the details of each parameter.

The plugin then generates the search space from the tuning parameters and the ATPs and selects a system configuration to explore. For each configuration, the plugin executes an experiment, which is an iteration of the phase region, and measures the objective values for the valid rts's of the significant regions. Each experiment first configures Score-P with the tuning parameter values for the phase region and the measurements for the specified objectives for the rts's.

After the execution of the phase, Score-P returns the measurements for the rts's, and Partial Calling Context Trees (CCT) * are generated at each analysis agents for the MPI processes controlled by it. The partial trees are then gathered to create the complete tree in the PTF frontend. The phase region represents the root node of the CCT, and its children represent call sites of the significant regions. Each child node can be viewed as an rts, ie, invocation of a significant region. If the region identifier is specified, an rts can be identified by its region name, the call path, and the region identifier. In Listing 1, a separate node representing an rts of `interpolate` is created for each grid level. Hence, the rts's of `interpolate` can now be distinguished in PTF via the call path, which includes the region identifiers (represented as *parameter_name=value*). If no region identifier is given, these rts's would be merged into a single node in the tree and would be indistinguishable for each grid level.

The measurements for the phase and the objective values for the rts's are then propagated to the tuning plugin. For the Energy objective, the plugin computes the consumed energy per experiment by aggregating the values returned by the designated processes of all the nodes for an MPI application. Since energy is a global metric of a node, only a designated single process in each node returns this value to Score-P. The plugin then outputs the best configuration for both the phase and the valid rts's. The plugin also performs additional experiments for verifying the savings incurred. The static saving for the phase is computed as the improvement in the energy consumption for the best setting of the tuning parameters over the energy consumed for the default setting of the tuning parameters. The static energy saving for the rts's is the improvement in the energy consumed by the rts's for the best static configuration for the phase over the energy consumed for the default configuration. The dynamic savings are computed as the improvement in the energy consumption for the best configuration for the rts's over the energy consumption for the best static configuration.

**RTS Database** The information about each rts, including its call path, region identifiers, default objective values, and the best and the worst setting of the tuning parameters for the READEX tuning plugin is stored in the RTS database.

**Tuning Model Generation** Finally, a tuning model is generated from the knowledge stored in the RTS database. The tuning model generation clusters the rts's into scenarios using a classifier. The classifier maps each valid rts onto a unique scenario based on the best configuration. The rts's may also be clustered based on similar best configurations using a similarity score that determines if the objective values are close to each other. Thus, instead of treating the two

* A context sensitive version of a call graph.

rts's as different scenarios, they can be merged into one scenario to reduce the overhead of scenario detection and switching at runtime. For each scenario, a selector is generated that returns the best configuration for that scenario with respect to the chosen objective. The tuning model encapsulates this knowledge as a JSON file.

**Tuning Model Merger** For each input identifier specified in the input specification file, a separate tuning model is generated at the end of DTA for the application. The Tuning Model Merger then merges all the tuning models into one merged application tuning model file, which is then read at runtime to perform dynamic switching.

**READEX Runtime Library (RRL)** The READEX Runtime Library (RRL) is used during both design-time and runtime. While RRL only switches the tuning parameter settings at design-time, it uses the ATP library and the merged application tuning model to look up the best configuration upon encountering a valid rts for each input identifier and ATP during production runs. If there is significant expected saving in the energy consumption over the switching overhead for an rts, the configuration is switched. RRL employs separate *parameter control plugins* for each tuning parameter to perform configuration switching.[18] RRL may also perform calibration by using different machine learning techniques to predict the optimal configuration if it encounters rts's that were not seen at design-time.

## 6 | EVALUATION

The evaluation of DTA using domain knowledge was performed on three applications: Multi-Grid (MG),[9] Block Tri-diagonal solver (BT-MZ),[9] and ESPRESO.[10,19,20] MG and BT-MZ, which are from the NAS Parallel Benchmarks suite, are derived from computational fluid dynamics (CFD) applications.[21] MG uses a V-cycle to solve a discrete Poisson equation on a sequence of meshes, with long and short distance communication, while switching between compute bound and memory bound at a specific grid level. The multi-zone version of BT uses uneven-size zones within one problem class and an increased number of zones with a bigger problem size. The ESPRESO library is a combination of Finite Element (FEM) tools and a domain decomposition-based Finite Element Tearing and Interconnect (FETI) solver. The FETI solver itself contains several variations of iterative solvers for which convergence can be improved by several preconditioners. The computational complexity of different preconditioners vary from basic vector scaling (weight function) to sparse-matrix vector multiplication with different number of non-zeros (lumped, light-Dirichlet) to dense matrix-vector multiplication (Dirichlet).

Experiments were conducted on the Taurus HPC system at the ZIH in Dresden. Each node on Taurus contains two 12-core Intel Xeon CPUs E5-2680 v3 (Intel Haswell family) running with a default CPU frequency of 2.50 GHz. Energy measurements are provided on the Taurus supercomputer via the HDEEM measurement hardware,[22] which allows processor as well as blade energy measurements.

First, too fine granular regions were filtered using a Score-P filter file, which contains all the regions that have to be eliminated from instrumentation. Then, `readex-dyn-detect` was run on the three applications separately, and significant regions for each application were output into a READEX configuration file. The phase region and the significant regions were annotated using Score-P macros (Section 4.1.1). The Energy objective was specified for all three applications.

The domain knowledge specification for MG was the grid level *k*, which was specified as the region identifier for the `interpolate` subroutine. For BT-MZ, two problem classes (B and C) were specified as the input identifiers. For ESPRESO, application tuning parameters, such as preconditioners, iterative solver methods, and domain decompositions were specified. DTA was performed for each application using the READEX tuning plugin, and the savings for the specified objective were computed. Finally, tuning models were generated for MG and ESPRESO, while a merged application tuning model was generated for BT-MZ.

### 6.1 | Region identifiers

Table 1 presents the results obtained from readex-dyn-detect and DTA for the MG benchmark after specifying the grid level as a region identifier. MG was executed for Class C with grid size 512x512x512 using 4 MPI processes and 1 OpenMP thread on a single node of the Taurus HPC system.

After applying readex-dyn-detect, the significant regions that were obtained are `rprj3`, `interp`, `psinv`, and `resid`. The valid rts's for each significant region of MG are presented in Table 1. The READEX tuning plugin executed a total of 70 experiments using the exhaustive search strategy. The best CPU frequency and uncore frequency setting for each valid rts of a significant region are shown in columns 5 and 6, respectively, along with the total energy consumed, as shown in column 3 and the execution time in column 4. As it can be seen, the best setting for the phase is not necessarily the best setting for all the rts's, and the best configuration varies for the rts's of `interp`. The static energy saving for the rts's amounts to 6.9% for all the significant regions and 4.5% for the phase region as compared to the default energy values. The improvement obtained using the READEX tuning for the rts's using domain knowledge was 2.2%.

Although there are both static and dynamic savings for MG, there is a trade-off between the energy consumption and the execution time.

**TABLE 1** Best configurations for valid runtime situations of significant regions of MG with grid level as a region identifier

| Region | Valid Rts's | Tuning Parameters | | | |
|---|---|---|---|---|---|
| | | Energy, J | Time, s | CPU_FREQ, GHz | UNCORE_FREQ, GHz |
| PhaseRegion | /PhaseRegion | 913.3 | 8.385 | 1.6 | 1.2 |
| rprj3 | /PhaseRegion/mg3P/rprj3 | 99.35 | 1.85 | 1.4 | 2.8 |
| interp | /PhaseRegion/mg3P/interp/level=2 | 11.76 | 0.111 | 1.2 | 2.8 |
| | /PhaseRegion/mg3P/interp/level=3 | 11.78 | 0.111 | 1.2 | 2.2 |
| | /PhaseRegion/mg3P/interp/level=4 | 11.77 | 0.112 | 1.2 | 1.6 |
| | /PhaseRegion/mg3P/interp/level=5 | 11.84 | 0.111 | 1.2 | 3.0 |
| | /PhaseRegion/mg3P/interp/level=6 | 11.89 | 0.112 | 1.2 | 1.6 |
| | /PhaseRegion/mg3P/interp/level=7 | 12.83 | 0.121 | 1.2 | 3.0 |
| | /PhaseRegion/mg3P/interp/level=8 | 27.56 | 0.224 | 2.4 | 1.2 |
| psinv | /PhaseRegion/mg3P/psinv | 132.1 | 2.05 | 2.2 | 1.4 |
| resid | /PhaseRegion/mg3P/resid | 126.45 | 3.48 | 2.0 | 1.8 |
| | /PhaseRegion/resid | 38.35 | 0.548 | 2.4 | 2.6 |

## 6.2 | Input identifiers

DTA was performed on BT-MZ with two problem classes, ie, B (64 zones) and C (256 zones), where each zone is assigned to a process for parallel execution. The two classes eventually are defined by zones, which have an uneven distribution of the workload across the X, Y, and Z dimensions. The READEX tuning plugin performed two different DTA runs for two input identifier files containing the specifications for Class B and Class C.

The READEX plugin executed 38 experiments with 8 MPI processes and 1 OpenMP thread on one node of the Taurus HPC system. Figures 5 and 6 show the trend in the energy consumption for 64 and 256 zones, respectively. The energy consumption in each of the graphs is presented for different settings of the CPU frequency and uncore frequency. As seen, for Class B, the trend in energy consumption slowly increases from a lower to a higher CPU frequency setting, while the energy consumption follows quite an irregular trend for Class C. This indicates that the work distribution of zones across processes is not the same. Thus, the best configuration for Class B is *{CPU_Freq=1.2, Uncore_Freq=1.0}* and Class C is *{CPU_Freq=1.3, Uncore_Freq=2.0}*.

Thus, an input identifier for the class size allows to distinguish the two situations in which different best configurations are obtained. This knowledge helps in the generation of a better tuning model by including information about the input identifier and thus returning the correct best configuration at runtime.

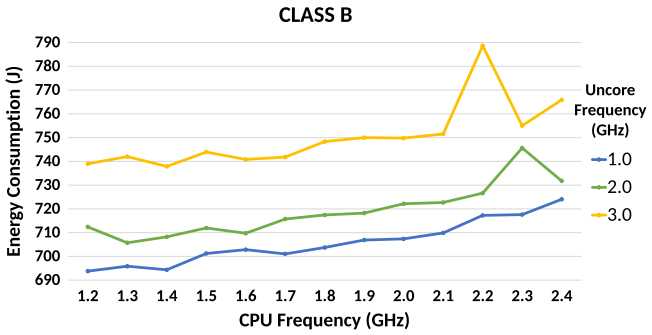


**FIGURE 5** Energy consumption of BT-MZ for Class B for different CPU frequency and uncore frequency settings during DTA

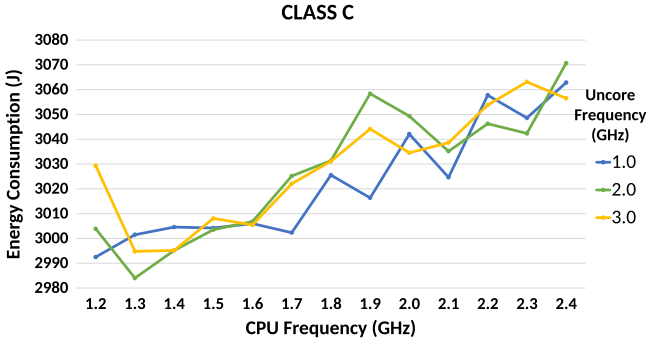


**FIGURE 6** Energy consumption of BT-MZ for Class C for different CPU frequency and uncore frequency settings during DTA

### 6.3 | Application Tuning Parameters

This section demonstrates the tuning potential and a comparison of the energy consumption obtained after using ATPs for ESPRESO.

Using a simplified approximation, we can state that, from the preconditioners listed in Section 6, the more computationally demanding the preconditioner is, the more numerically efficient it is, ie, the more it reduces the number of iterations to solve the problem. In ESPRESO, we can dynamically switch between any of these during the runtime. If a preconditioner is not used, one iteration contains an action of a FETI operator (cost is 30.9 J and 0.12 s) and an application of a projector (cost is 0.7 J and 0.005 s). If a preconditioner is used, each iteration contains one more projector application in addition to the preconditioner action.

We evaluated a projected conjugate gradient (PCG) solver with the preconditioners on a structural mechanics (linear elasticity) problem with 2.3 million unknowns on a single compute node using 24 MPI processes. The results in Table 2 show that the solution can be reached in 5.46 s when using Light Dirichlet preconditioner, despite the fact that it needs more iterations than the Dirichlet preconditioner. The Light Dirichlet preconditioner saved 15.9 s and 4 091.5 J in comparison to solving the problem without any preconditioner.

ESPRESO can switch during the runtime not only the preconditioners but also the iterative solver methods. The default Projected CG (PCG) solver can be replaced by a pipelined version (PIPEPCG), a fully orthogonal (ORTH_PCG) version, or a fully orthogonal version with restarts (ORTH_PCG_CP) of the Conjugate Gradient (CG) algorithm. Other available solvers use a biconjugate gradient stabilized method (BiCGSTAB) or generalized minimal residual method (GMRES) are used for non-symmetric linear systems.

We evaluated the different linear solvers on a heat transfer problem with 14.2 million unknowns. The tests ran on a single node with two MPI processes, each with 12 OpenMP threads per process. This maps to the dual socket node architecture where each CPU has 12 cores. The results are presented in Table 3. The ORTH_PCG_CP solver was able to solve the problem in 66.1 seconds instead of 94 seconds (default PCG solver) and save 25.8 % (6700 J) of the overall energy.

The same problem with 14.2 million unknowns was also evaluated for different domain decompositions. ESPRESO uses METIS library[23] for problem partitioning. The overall problem was decomposed into $24 * 2^n$ domains for $n$ in 0..6, as shown in the Table 4. With rising number of domains (and decreasing domain size accordingly), the consumed energy and time dropped down, until 192 elements per domain was reached. At this point, the runtime was shorter by about 16 %, and the consumed energy decreased by about 22.7 % (5844.9 J saved). If smaller domains were used, both runtime and energy consumption grew rapidly.

Runtime switching can be connected with extra expenses. Almost no overhead is associated with switching the preconditioners and iterative solvers. On the other hand, switching the decomposition size requires expensive preprocessing and therefore, it is mostly suitable for static tuning or transient problems with a high number of time steps.

The presented results show good trends and savings but are not generally applicable to every problem that can be solved by ESPRESO. Different physics, different problem sizes, and material properties will affect the savings that can be achieved. For instance, the domain decomposition will play a more significant role for larger problems.

**TABLE 2** Comparison of the ESPRESO preconditioners with respect to energy consumption and runtime. The table contains (i) single iteration evaluation including baseline (FETI operator and 2x projector) + resources spent by the preconditioner and (ii) overall FETI solver evaluation considering the different number of solver iterations

| Preconditioner | # iterations | 1 iteration | | Solution | |
|---|---|---|---|---|---|
| **none** | 172 | 125 ms | 31.6 J | 21.36 s | 5 501.31 J |
| **Weight function** | 100 | 130+2 ms | 32.3+0.53 J | 12.89 s | 3 284.07 J |
| **Lumped** | 45 | 130+10 ms | 32.3+3.86 J | 6.32 s | 1 636.11 J |
| **Light Dirichlet** | 39 | 130+10 ms | 32.3+3.74 J | 5.46 s | 1 409.82 J |
| **Dirichlet** | 30 | 130+80 ms | 32.3+20.62 J | 6.34 s | 1 594.50 J |

**TABLE 3** Comparison of the ESPRESO solvers with respect to energy consumption and time

| | PCG | BiCGSTAB | GMRES | ORTH_PCG | ORTH_PCG_CP | PIPEPCG |
|---|---|---|---|---|---|---|
| **Energy [J]** | 26070.0 | 26585.8 | 26239.0 | 26056.5 | 19341.2 | 26256.5 |
| **Time [s]** | 94.00 | 98.38 | 94.49 | 93.88 | 66.07 | 94.49 |

**TABLE 4** Comparison of energy and time consumed to reach the problem solution if the problem was split into different number of domains

| | 24 | 48 | 96 | 192 | 384 | 768 | 1536 |
|---|---|---|---|---|---|---|---|
| **Energy [J]** | 25707.6 | 22057.1 | 20737.9 | 19862.7 | 19969.8 | 22046.9 | 23389.7 |
| **Time [s]** | 98.92 | 87.80 | 84.23 | 82.17 | 83.67 | 92.86 | 99.42 |

## 7 | SUMMARY

This paper first presents the domain knowledge specification for the application structure, namely, the region and phase specifications. The specification of a user region enables to aggregate too fine granular regions for intra-phase tuning, ie, dynamism existing between rts's within the phase. The phase specification allows to explore inter-phase dynamism, ie, dynamism between phases in addition to the intra-phase dynamism of the application. Region identifiers are used to define specific characteristics of a computation and can be used to enhance the tuning model by distinguishing more rts's. Input identifiers characterize different input behaviors and enable DTA to generate a global merged application tuning model. The Domain Knowledge Specification Interface also enables the application expert to specify new tuning parameters via the ATP library. DTA then takes these tuning parameters into account and creates a more extensive search space for energy tuning. This paper demonstrates a 2.2% dynamic energy saving using region identifiers for the MG benchmark due to DTA being able to distinguish more rts's than usual. Trends in the energy consumption for two different input identifiers specified by Class B and Class C for the BT-MZ benchmark are also discussed. Finally, the performance improvements are assessed with respect to energy consumption and runtime for ESPRESO by using different application tuning parameters such as preconditioners, iterative solver methods, and domain decompositions. For the preconditioner ATP, the Light Dirichlet preconditioner shows a 74.37% saving in energy consumption with respect to other preconditioners. Similarly, using the ORTH_PCG_CP solver method and the 192 elements per domain, the application expert saved 25.8% and 22.7% energy in comparison to the default (PCG) solver and 24 elements per domain, respectively.

## 8 | CONCLUSIONS

As of today, the major challenge in HPC research is saving energy consumption for Exascale computing. This paper has given an overview of the READEX project, which is aiming at improving the energy efficiency of HPC applications by a dynamic tuning approach. Furthermore, it emphasizes that the use of the domain knowledge specification approach introduced in the READEX methodology in fact results in dynamic savings. This results in an enhanced tuning model in addition to the auto-tuning approach in order to guide the dynamic switching of the tuning parameters. The application owners can provide their expert knowledge about the application domain by specifying *identifiers* and application-level tuning parameters that significantly increase the tuning potential. We present some preliminary results of different domain knowledge aspects and evaluate their impact on the tuning process.

### ACKNOWLEDGMENT

The research leading to these results has received funding from the European Union's Horizon 2020 Programme under grant 671657.

### ORCID

*Madhura Kumaraswamy* http://orcid.org/0000-0002-4817-5403

### REFERENCES

1. Bates NJ, Patterson MK. Achieving the 20 MW target: Mobilizing the HPC community to accelerate energy efficient computing. In: D'Hollander E, Dongarra JJ, Foster IT, eds. *Transition of HPC Towards Exascale Computing. Advances in Parallel Computing*. Amsterdam, The Netherlands: IOS Press; 2013:37-45.
2. Oleynik Y, Gerndt M, Schuchart J, Kjeldsberg PG, Nagel WE. Run-time exploitation of application dynamism for energy-efficient exascale computing (READEX). Paper presented at: Plessl C, El Baz D, Cong G, Cardoso JMP, Veiga L, Rauber T, eds. Paper presented at: 2015 IEEE 18th International Conference on Computational Science and Engineering CSE; 2015; Porto, Portugal. https://doi.org/10.1109/CSE.2015.55
3. Guillen C, Navarrete C, Brayford D, Hesse W, Brehm M. DVFS automatic tuning plugin for energy related tuning objectives. Paper presented at: 2016 2nd International Conference on Green High Performance Computing ICGHPC; 2016; Nagercoil, India. https://doi.org/10.1109/ICGHPC.2016.7508061
4. Hackenberg D, Schöne R, Ilsche T, Molka D, Schuchart J, Geyer R. An energy efficiency feature survey of the Intel Haswell processor. Paper presented at: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. 2015; Hyderabad, India. https://doi.org/10.1109/IPDPSW.2015.70
5. Schuchart J, Gerndt M, Kjeldsberg PG, et al. The READEX formalism for automatic tuning for energy efficiency. *Computing*. 2017;99(8):727-745. https://doi.org/10.1007/s00607-016-0532-7
6. Gerndt M, César E, Benkner S. *Automatic Tuning of HPC Applications - The Periscope Tuning Framework*; 2015; Herzogenrath, Germany: Shaker Verlag. ISBN: 978-3-8440-3517-9.
7. Knüpfer A, Rössel C, an Mey D, et al. Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Brunst H, Müller M, Nagel WE, Resch MM, eds. *Tools for High Performance Computing 2011*. Berlin, Germany: Springer; 2012:79-91.
8. Score-E: Scalable Tools for the Analysis and Optimization of Energy Consumption in HPC. http://www.vi-hps.org/projects/score-e
9. Bailey DH. The NAS parallel benchmarks. In: Padua D, ed. *Encyclopedia of Parallel Computing*. New York, NY: Springer; 2011:1254-1259.
10. Říha L, Brzobohatý T, Markopoulos A, Meca O, Kozubek T. Massively parallel hybrid total FETI (HTFETI) solver. In: PASC'16 Proceedings of the Platform for Advanced Scientific Computing Conference; 2016; Lausanne, Switzerland. https://doi.org/10.1145/2929908.2929909
11. Navarrete CB, Guillén C, Hesse W, Brehm M. Autotuning the energy consumption. In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. Amsterdam, The Netherlands: IOS Press; 2016:668-677. https://doi.org/10.3233/978-1-61499-381-0-668

12. Morajko A, Margalef T, Luque E. Design and implementation of a dynamic tuning environment. *J Parallel Distrib Comput*. 2007;67(4):474-490. https://doi.org/10.1016/j.jpdc.2007.01.001
13. Calore E, Gabbana A, Schifano SF, Tripiccione R. Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications. *Concurrency Computat Pract Exper*. 2017;29(12). https://doi.org/10.1002/cpe.4143
14. Silvano C, Agosta G, Cherubin S, et al. The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems. In: Proceedings of the ACM International Conference on Computing Frontiers. ACM; 2016:288-293. https://doi.org/10.1145/2903150.2903470
15. Highly accurate finite element simulation for sheet metal forming. http://gns-mbh.com/en/produkte/indeed/
16. Rosser E, Kelly W, Pugh B, Wonnacott D, Shpeisman T, Maslov V. The Omega project. http://www.cs.umd.edu/projects/omega/
17. Rosser E, Kelly W, Pugh B, Wonnacott D, Shpeisman T, Maslov V. The Omega calculator. https://github.com/davewathaverford/the-omega-project
18. Kjeldsberg PG, Gocht A, Gerndt M, Riha L, Schuchart J, Mian US. READEX: Linking two ends of the computing continuum to improve energy-efficiency in dynamic applications. In: 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE); 2017; Lausanne, Switzerland. https://doi.org/10.23919/DATE.2017.7926967
19. Říha L, Brzobohatý T, Markopoulos A, Meca O, Merta M. ESPRESO - ExaScale PaRallel FETI SOlver. http://espreso.it4i.cz. Accessed January 14, 2016.
20. Říha L, Brzobohatý T, Markopoulos A. Hybrid parallelization of the total FETI solver. *Adv Eng Softw*. 2017:103:29-37. https://doi.org/10.1016/j.advengsoft.2016.04.004
21. Haoqiang J, Van der Wijngaart RF. Performance characteristics of the multi-zone NAS parallel benchmarks. *J Parallel Distributed Comput*. 2006;66(5):674-685.
22. Hackenberg D, Ilsche T, Schuchart J, et al. HDEEM: High Definition Energy Efficiency Monitoring. In: Energy Efficient Supercomputing Workshop E2SC; 2014; New Orleans, LA. https://doi.org/10.1109/E2SC.2014.13
23. Karypis G, Kumar V. Metis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. 2009. http://www.cs.umn.edu/~metis

**How to cite this article:** Kumaraswamy M, Chowdhury A, Gerndt M, et al. Domain knowledge specification for energy tuning. *Concurrency Computat Pract Exper*. 2019;31:e4650. https://doi.org/10.1002/cpe.4650